Tripartite Modular Multiplication

Kazuo Sakiyama[1,2], Miroslav Knežević[1], Junfeng Fan[1], Bart Preneel[1], and Ingrid Verbauwhede[1]

[1] Katholieke Universiteit Leuven
Department of Electrical Engineering ESAT/SCD-COSIC
and
Interdisciplinary Center for Broad Band Technologies
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{miroslav.knezevic,junfeng.fan,bart.preneel,ingrid.verbauwhede}@esat.kuleuven.be
[2] The University of Electro-Communications
Department of Information and Communication Engineering
1-5-1 Chofugaoka, Chofu, Tokyo 182-8585, Japan
saki@ice.uec.ac.jp

Abstract. This paper presents a new modular multiplication algorithm that allows one to implement modular multiplications efficiently. It proposes a systematic approach for maximizing a level of parallelism when performing a modular multiplication. The proposed algorithm effectively integrates three different existing algorithms, a classical modular multiplication based on Barrett reduction, the modular multiplication with Montgomery reduction and the Karatsuba multiplication algorithms in order to reduce the computational complexity and increase the potential of parallel processing. The algorithm is suitable for both hardware implementations and software implementations in a multiprocessor environment. To show the effectiveness of the proposed algorithm, we implement several hardware modular multipliers and compare the area and performance results. We show that a modular multiplier using the proposed algorithm achieves a higher speed comparing to the modular multipliers based on the previously proposed algorithms.

Keywords: Modular multiplication, Barrett reduction, Karatsuba multiplication, Montgomery reduction, public-key cryptography

1 Introduction

Modular multiplication is a mathematical operation that is most commonly used in Public-Key Cryptography (PKC), *e.g.* RSA [11] and Elliptic Curve Cryptography (ECC) [7, 8]. RSA modular exponentiation and Elliptic Curve (EC) point multiplication are performed by using modular multiplications repeatedly. Namely, modular multiplication is an essential building block that determines the performance of RSA and ECC in terms of cost and speed. Implementing an efficient modular multiplication for PKC has been a great challenge for both software and hardware platforms because one has to deal with large numbers or polynomials, *i.e.* at least 1024-bit integer for RSA and 160 bits or more for ECC.

Given two n-digit integers A and B, a modular multiplication algorithm returns $C = AB \bmod M$, where M is an n-digit modulus and $M > A, B$. The multiplication ($C' = AB$) and the reduction ($C = C' \bmod M$) can be separated or interleaved. Modular multiplication can be sped up by complexity reduction and parallel computing.

For example, using Karatsuba's method [6] reduces the number of sub-word multiplications, while using bipartite multiplication algorithm [5] enables a two-way parallelism.

Complexity reduction On the algorithmic level, complexity of the modular operations can be reduced by smart integer multiplication methods or fast reduction methods. For example, long integer multiplication has complexity of $O(n^2)$, where n is the number of digit size. When using Karatsuba's algorithm, the complexity can be reduced down to $O(n^{log_2 3})$. The complexity of the reduction phase can be significantly reduced if the modulus is a pseudo-Mersenne number. However, the use of pseudo-Mersenne numbers is applicable only to a certain number of cryptosystems today.

Parallelization Parallel computing increases the speed at the cost of larger area. Many popular platforms today incorporate multiple Arithmetic Logic Units (ALU) or even processors. For example, the latest FPGAs have multiple embedded multipliers or DSP slices, making parallel computing possible and attractive. The main challenge of parallelizing an algorithm lies in the task partitioning and memory management. When targeting a multicore platform, an algorithm is normally tweaked such that it is more

friendly to parallel implementations [4]. In another instance, the Montgomery modular multiplication was tweaked and implemented on FPGA with multiple DSP slices [12].

This paper proposes a new modular multiplication algorithm that effectively integrates three existing algorithms, a classical modular multiplication based on Barrett reduction, the Montgomery modular multiplication and the Karatsuba multiplication. The novelty comes at higher algorithmic level and can be further optimized by parallelizing any of its ingredients (Barrett, Montgomery or Karatsuba multiplication). The proposed algorithm minimizes the number of single-precision (SP) multiplications and enables more than 3-way parallel computation. This paper investigates the cost and speed trade-off for a hardware implementation of the proposed modular multiplication algorithm and compares the results with implementations of previous algorithms.

The remainder of this paper is structured as follows. Section 2 describes previous work and outlines basics, which are necessary for further discussion. In Section 3, our proposed algorithm is introduced. Section 3 also discusses the trade-off between cost and speed of our proposed algorithm theoretically, compared with existing algorithms. Several hardware implementation results are introduced for various implementation options in Section 4. Section 5 concludes the paper.

2 Related Work

In the paper we use the following notations. A multiple-precision n-bit integer A is represented in radix r representation as $A = (A_{n_w-1} \dots A_0)_r$ where $r = 2^w$; n_w represents the number of words and is equal to $\lceil n/w \rceil$ where w is a *word-size*; A_i is called a *digit* and $A_i = (a_{w-1} \dots a_0) \in [0, r-1]$. A special case is when $r = 2$ ($w = 1$) and the representation of $A = (a_{n-1} \dots a_0)_2$ is called a *bit* representation. A multiplication of two digits is further referred to as a *single-precision* (SP) multiplication. Sometimes we refer to it as a *digit* multiplication. We define a *partial product* as a product of a single digit and an n_w-digit integer.

Given a modulus M and two elements $A, B \in \mathbb{Z}_M$ where $\mathbb{Z}_M$ is the ring of integers modulo M, we define the ordinary modular multiplication as:

$$A \times B \triangleq AB \bmod M \ . \tag{1}$$

Two algorithms for efficient modular reduction, namely Barrett [2] and Montgomery [9] algorithms are widely used today. Both algorithms avoid multiple-precision division, which is considered expensive. The classical modular multiplication algorithm, based on Barrett's reduction, uses single-precision multiplications with the precomputed modulus reciprocal instead of expensive divisions. The algorithm that efficiently combines classical and Montgomery multiplications in finite fields of characteristic 2 was independently proposed by Potgieter [10] and Wu [14] in 2002. Published in 2005, a bipartite modular multiplication by Kaihara and Takagi [5] extended this approach to the ring of integers.

2.1 Barrett reduction

The classical modular multiplication algorithm computes $AB \bmod M$ by interleaving the multiplication and modular reduction phases as it is shown in Alg. 1. The calculation of the intermediate quotient q_C at

Algorithm 1 Classical modular multiplication algorithm.

Input: $A = (A_{n_w-1} \dots A_0)_r$, $B = (B_{n_w-1} \dots B_0)_r$, $M = (M_{n_w-1} \dots M_0)_r$ where $0 \le A, B < M$, $r^{n_w-1} \le M < r^{n_w}$ and $r = 2^w$.
Output: $T = AB \bmod M$.
1: $T \Leftarrow 0$
2: **for** $i = n_w - 1$ downto 0 **do**
3: $\quad T \Leftarrow Tr + AB_i$
4: $\quad q_C = \lfloor T/M \rfloor$
5: $\quad T \Leftarrow T - q_C M$
6: **end for**
7: Return T.

step 4 of the algorithm is done by utilizing integer division which is considered as an expensive operation.

The idea of using the precomputed reciprocal of the modulus M and simple shift and multiplication operations instead of division was first introduced by Barrett in his master thesis [1]. To explain the basic idea, we rewrite the intermediate quotient q_C as:

$$q_C = \left\lfloor \frac{T}{M} \right\rfloor = \left\lfloor \frac{\frac{T}{2^{n+\beta}} \frac{2^{n+\alpha}}{M}}{2^{\alpha-\beta}} \right\rfloor \geq \left\lfloor \frac{\left\lfloor \frac{T}{2^{n+\beta}} \right\rfloor \left\lfloor \frac{2^{n+\alpha}}{M} \right\rfloor}{2^{\alpha-\beta}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{T}{2^{n+\beta}} \right\rfloor \mu}{2^{\alpha-\beta}} \right\rfloor = \hat{q} \ . \tag{2}$$

The value $\hat{q}$ represents an estimation of the intermediate quotient q_C. In most of the cryptographic applications, the modulus M is fixed during the many modular multiplications and hence the value $\mu = \lfloor 2^{n+\alpha}/M \rfloor$ can be precomputed and reused multiple times. Since the value of $\hat{q}$ is an estimated value, some correction steps at the end of the modular multiplication algorithm have to be performed. In his thesis, Dhem [3] determines the values of $\alpha = w + 3$ and $\beta = -2$ for which the classical modular multiplication based on Barrett reduction, needs at most one subtraction at the end of the algorithm. The only drawback of the proposed method is the size of the intermediate quotient q_C and the precomputed value μ. Due to the parameters α and β chosen in a given way, the size of q_C is $w + 2$ and μ is at most $w + 4$ bits. This introduces an additional overhead for software implementations, while it can be easily overcome in hardware.

In what follows, we shall often refer to the classical modular multiplication based on Barrett reduction simply as Barrett multiplication.

2.2 Montgomery reduction

Montgomery's algorithm [9] is the most commonly utilized modular multiplication algorithm today. In contrast to the classical modular multiplication, it utilizes right to left divisions. Given an n-digit odd modulus M and an integer $U \in \mathbb{Z}_M$, the image or the Montgomery residue of U is defined as $X = UR \bmod M$ where R, the Montgomery radix, is a constant relatively prime to M. If A and B are, respectively, the images of U and V, the Montgomery multiplication of these two images is defined as:

$$A * B \triangleq ABR^{-1} \bmod M \ . \tag{3}$$

The result is the image of $UV \bmod M$ and needs to be converted back at the end of the process. For the sake of efficiency, one usually uses $R = r^n$ where $r = 2^w$ is the radix of each digit. Similar to Barrett multiplication, this algorithm uses a precomputed value $M' = -M^{-1} \bmod r = -M_0^{-1} \bmod r$. The algorithm is shown in Alg. 2.

Algorithm 2 Montgomery modular multiplication algorithm.

Input: $A = (A_{n_w-1} \dots A_0)_r$, $B = (B_{n_w-1} \dots B_0)_r$, $M = (M_{n_w-1} \dots M_0)_r$, $M' = -M_0^{-1} \bmod r$ where $0 \leq A, B < M$, $2^{n_w-1} \leq M < 2^{n_w}$, $r = 2^w$ and $\gcd(M, r)$=1.
Output: $T = ABr^{-n_w} \bmod M$.
1: $T \Leftarrow 0$
2: **for** $i = 0$ to $n_w - 1$ **do**
3: $\quad T \Leftarrow T + AB_i$
4: $\quad q_M \Leftarrow (T \bmod r)M' \bmod r$
5: $\quad T \Leftarrow (T + q_M M)/r$
6: **end for**
7: **if** $T \geq M$ **then**
8: $\quad T \Leftarrow T - M$
9: **end if**
10: Return T.

2.3 Bipartite Modular Multiplication

The bipartite algorithm was introduced for the purpose of a two-way parallel computation [5]. It uses two custom modular multipliers, a classical modular multiplier and a Montgomery multiplier, in order to improve the speed. By combining a classical modular multiplication with Montgomery's modular

multiplication, it splits the operand multiplier into two parts and processes them in parallel, increasing the calculation speed. The calculation is performed using Montgomery residues defined by a modulus M and a Montgomery radix R, $R < M$. Next, we outline the main idea of the bipartite algorithm.

Let $R = r^k$ for some $0 < k < n_w$. Consider the multiplier B to be split into two parts B_1 and B_0 so that $B = B_1R + B_0$. Then, the Montgomery multiplication modulo M of the integers A and B can be computed as follows:

$$\begin{aligned} A * B &= ABR^{-1} \bmod M \\ &= A(B_1R + B_0)R^{-1} \bmod M \\ &= \big((AB_1 \bmod M) + (AB_0R^{-1} \bmod M)\big) \bmod M \ . \end{aligned} \tag{4}$$

The left term of the last equation, $AB_1 \bmod M$, can be calculated using the classical modular multiplication that processes the upper part of the split multiplier B_1. The right term, $AB_0R^{-1} \bmod M$, can be calculated using the Montgomery algorithm that processes the lower part of the split multiplier B_0. Both calculations can be performed in parallel. Since the split operands B_1 and B_0 are shorter in length than B, the calculations $AB_1 \bmod M$ and $AB_0R^{-1} \bmod M$ are performed faster than $ABR^{-1} \bmod M$.

3 The Proposed Modular Multiplication Algorithm

Here, we introduce our new modular multiplication algorithm that achieves a higher speed comparing to the bipartite algorithm. The algorithm is very suitable for the multicore platforms where an ample parallelism provided by the tripartite algorithm can be exploited.

Among the integer multiplication techniques, two important methods are Karatsuba algorithm [6] and its generalization – Toom-Cook's algorithm (sometimes known as Toom-3) [13]. They both reduce the number of single-precision multiplications by reusing the intermediate partial products. By recursively using Karatsuba's method, one multiplication of two n_w-digit integers has complexity of $\mathcal{O}(n_w^{\log_2 3})$, while Toom-$k$ has complexity $\mathcal{O}(n_w^{\log_k 2k-1})$. Both algorithms can provide faster multiplication than the normal schoolbook method. Karatsuba's algorithm can accelerate multiplication by representing two n_w-digit integers as $A = A_1R + A_0$ and $B = B_1R + B_0$, where $R = 2^k$ is chosen for an efficient implementation. Then, the product of AB can be computed as:

$$AB = p_1R^2 + (p_2 - p_0 - p_1)R + p_0 \ , \tag{5}$$

where

$$p_0 = A_0B_0,\ p_1 = A_1B_1,\ p_2 = (A_0 + A_1)(B_0 + B_1) \ . \tag{6}$$

Therefore, we need only three sub-product multiplications while the standard, schoolbook multiplication needs four sub-products. The highest speed is achieved when choosing k to be around $n_w/2$. By using Karatsuba's method recursively, the time complexity becomes $\mathcal{O}(n_w^{\log_2 3})$.

3.1 Overview of the Proposed Multiplication Algorithm

We explain the proposed algorithm by starting from the basic version that is based on the following equation derived from Karatsuba's algorithm.

$$\begin{aligned} ABR^{-1} \bmod M =& (A_1R + A_0)(B_1R + B_0)R^{-1} \bmod M \\ =& \big\{A_1B_1R + (A_1B_0 + A_0B_1) + A_0B_0R^{-1}\big\} \bmod M \\ =& \big\{p_1R \bmod M \\ &+ (p_2 - p_0 - p_1) \bmod M \\ &+ p_0R^{-1} \bmod M\big\} \bmod M \ , \end{aligned} \tag{7}$$

where

$$p_0 = A_0B_0,\ p_1 = A_1B_1,\ p_2 = (A_0 + A_1)(B_0 + B_1) \ . \tag{8}$$

In Eq. (7), n_w-digit inputs A and B are split into two blocks as $A = (A_1, A_0)_R$ and $B = (B_1, B_0)_R$, and then Karatsuba's method is applied for performing multiplication of AB. Here, R is chosen as $R = r^k$ where $k = \lceil n_w/2 \rceil$ for an efficient implementation although k can be arbitrarily chosen. We call this case

a u-split version where $u = 2$. In total, we have three terms that can be computed independently by using the existing algorithms described in the previous sections. To obtain a high-speed implementation, one can compute these three different terms in parallel. Figure 1 explains the main idea of the proposed algorithm and compares with the bipartite algorithm.

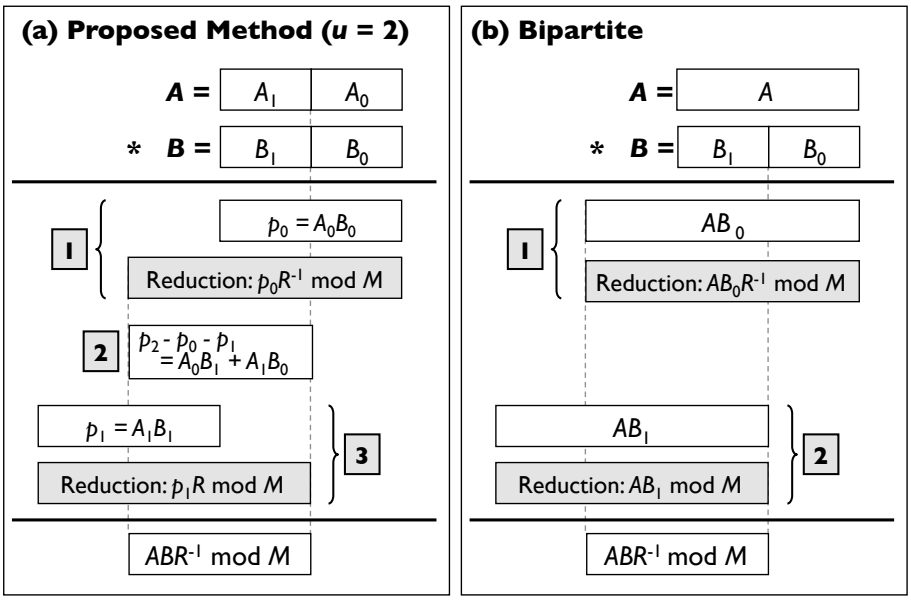

Fig. 1. Procedure for modular multiplication. (a) our proposed method. (b) bipartite method.

3.2 Further Exploration of the Proposed Algorithm

For a further exploration of the proposed algorithm, we can split the inputs into more blocks, e.g. $A = (A_3, A_2, A_1, A_0)_R$ for an n_w-digit integer where $R = r^k$ and $k = \lceil n_w/4 \rceil$. In this example case of $u = 4$, we can explore further parallelism as

$$\begin{aligned} ABR^{-2} \bmod M = \Big[& p_3 R^4 \bmod M + (p_7 - p_2 - p_3) R^3 \bmod M \\ & + \big\{ (p_6 - p_1 + p_2 - p_3) R^2 \\ & + (p_8 + p_0 + p_1 + p_2 + p_3 - p_4 - p_5 - p_6 - p_7) R \\ & + (p_5 - p_0 + p_1 - p_2) \big\} \bmod M \\ & + (p_4 - p_0 - p_1) R^{-1} \bmod M + p_0 R^{-2} \bmod M \Big] \bmod M \ , \end{aligned} \tag{9}$$

where

$$\begin{aligned} & p_0 = A_0 B_0, \ p_1 = A_1 B_1, \ p_2 = A_2 B_2, \ p_3 = A_3 B_3 \ , \\ & p_4 = (A_0 + A_1)(B_0 + B_1), \ p_5 = (A_0 + A_2)(B_0 + B_2) \ , \\ & p_6 = (A_1 + A_3)(B_1 + B_3), \ p_7 = (A_2 + A_3)(B_2 + B_3) \ , \\ & p_8 = (A_0 + A_1 + A_2 + A_3)(B_0 + B_1 + B_2 + B_3) \ . \end{aligned} \tag{10}$$

This example case allows us to perform modular multiplication up to 5-way parallel computing as shown in Eq. (9). Parameters $p_0, p_1, \ldots, p_8$ in Eq. (10) are computed with complexity of 9 sub-product multiplications and 14 additions.

For the reduction steps, we apply Barrett and Montgomery reduction to the terms that require reduction (i.e. 1st, 2nd, 6th and 7th term in Eq. (9)). For the other terms, we use a simple modular addition or subtraction. The final reduction step is performed after adding up all the partial results derived from each term.

Due to the carry-free arithmetic, for a (modular) multiplication over a binary field, the reduction is only needed for terms that require Barrett and Montgomery reduction (again, these are 1st, 2nd, 6th and 7th term in Eq. (9)). Figure 2 a summarizes the 4-split version of the proposed algorithm.

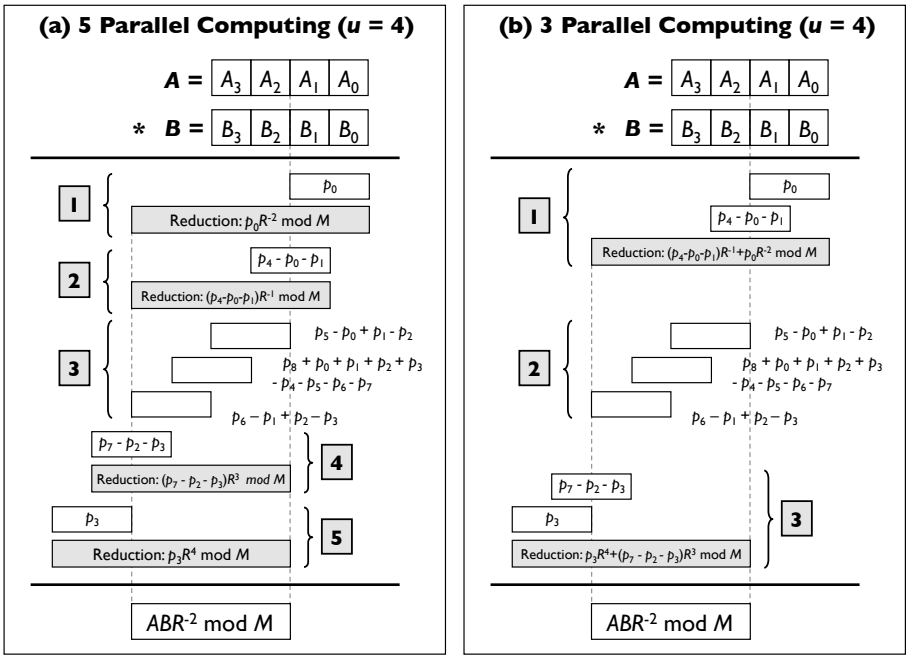

Fig. 2. Procedure for modular multiplication for $u = 4$. (a) five-way parallel computation. (b) three-way parallel computation.

For a better area-performance trade-off, the method described in Fig. 2 (a) can be modified as shown in Fig. 2 (b). It illustrates a 3-way parallel computational sequence equivalent to the one in Fig. 2 a. The 3-way parallel version can save area cost by sharing the hardware for reduction. The critical path delay increases slightly due to the one extra addition before the reduction. However, this speed penalty occurs only for the case of integer multiplication where the carry propagation is present.

To illustrate further, we also mention the case of $u = 8$ for which we need 27 $\lceil n_w/8 \rceil \times \lceil n_w/8 \rceil$ digit multiplications to prepare the partial products $p_0, p_1, \ldots, p_{26}$ since each $\lceil n_w/4 \rceil \times \lceil n_w/4 \rceil$ digit multiplication in Eq. (10) can be computed with three $\lceil n_w/8 \rceil \times \lceil n_w/8 \rceil$ digit by using Karatsuba's method. In general, we need 3^v $\lceil n_w/u \rceil \times \lceil n_w/u \rceil$ digit multiplications for $u = 2^v$, where v is a non-negative integer.

Finally, to give an idea of possible improvements of our algorithm, we provide Fig. 3 which represents the hierarchy of modular multiplication. In order to further improve the performance of tripartite (and bipartite) multiplication, one can use an optimized (pipelined, parallelized, etc) implementation of any of the levels below. For example, the pipelined implementation of Montgomery multiplication, as described by Suzuki [12], can be used to further parallelize our algorithm. However, in this section we focus on the higher algorithmic level and therefore we leave this practical question open for further analysis.

4 Cost and Performance Estimation

As explained above, our proposed algorithm highly benefits from using parallel computations and we consider two possible scenarios for the choice of our hardware architecture. First, we consider a fully-parallel implementation, an approach where the modular multiplication is performed very fast, within a single clock cycle. Although the fully-parallel implementation provides a very fast modular multiplication, it comes at large area overhead. Second, we discuss a trade-off, meaning a digit-serial approach that provides a reasonably high speed, yet maintaing a relatively low area. For the sake of our discussion, we observe the case of two-split version ($u = 2$) and use Fig. 1a as a reference.

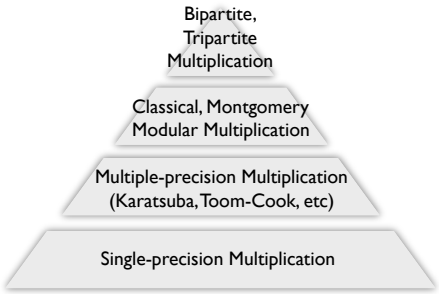

Fig. 3. Hierarchy of the modular multiplication.

Additionally, to prove the superiority of our algorithm, we provide hardware implementations and compare them with the architectures based on Barrett, Montgomery, and bipartite algorithms. Note here that we do not attempt to provide highly optimized implementations by any means. Our hardware implementations rather serve as a proof of concept and make sure that the comparison with other algorithms is done using the same framework. Further improvements of our implementations are certainly possible by optimizing at lower levels of abstractions.

Fully-Parallel Implementation By the means of fully-parallel implementation we consider a very fast, one-clock-cycle modular multiplier. In order to execute Barrett or Montgomery multiplication in a single clock cycle, we need three $n \times n$-bit multipliers, which is equivalent to the size of twelve $\frac{n}{2} \times \frac{n}{2}$-bit multipliers. The fully-parallel architectures supporting Barrett and Montgomery algorithms are illustrated in Fig. 4a and Fig. 4b, respectively. We implement $n = 192$-bit modular multipliers in both cases.

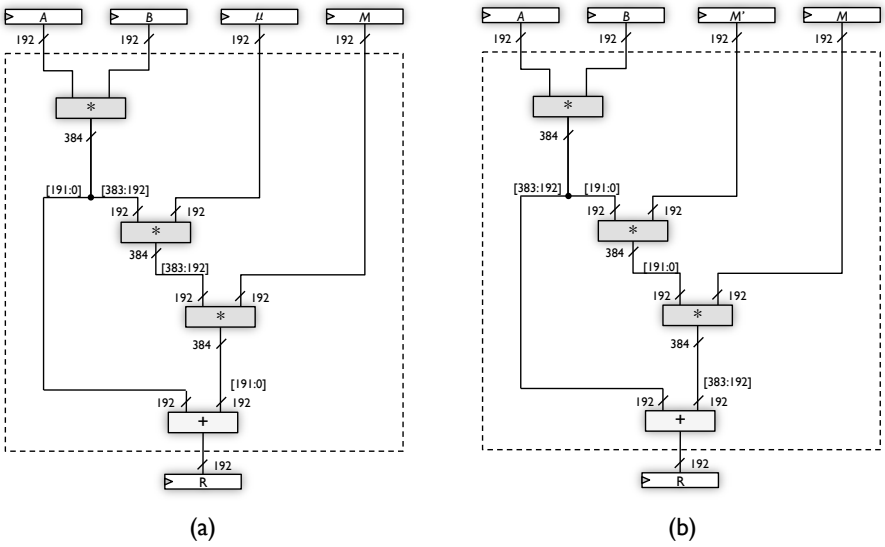

Fig. 4. Datapath of a fully-parallel modular multiplier based on (a) classical and (b) Montgomery algorithm.

In order to achieve the same speed, a bipartite modular multiplier needs three $n \times \frac{n}{2}$-bit multipliers to calculate $AB_0R^{-1} \bmod M$ and another three $n \times \frac{n}{2}$-bit multipliers to calculate $AB_1R \bmod M$. In total, we need six $n \times \frac{n}{2}$-bit multipliers, which size is equivalent to the size of twelve $\frac{n}{2} \times \frac{n}{2}$-bit multipliers. However, the bipartite multiplier can be implemented more efficiently using only ten $\frac{n}{2} \times \frac{n}{2}$-bit multipliers. This is illustrated in Fig. 5, where the implementation of $n = 192$-bit modular multiplier is given.

Finally, we implement our proposed algorithm in a fully-parallel manner. Using the notations from Fig. 1, we perform $p_0R^{-1} \bmod M$, $p_1R \bmod M$, and $(p_2 - p_0 - p_1) \bmod M$ in a single clock cycle, all

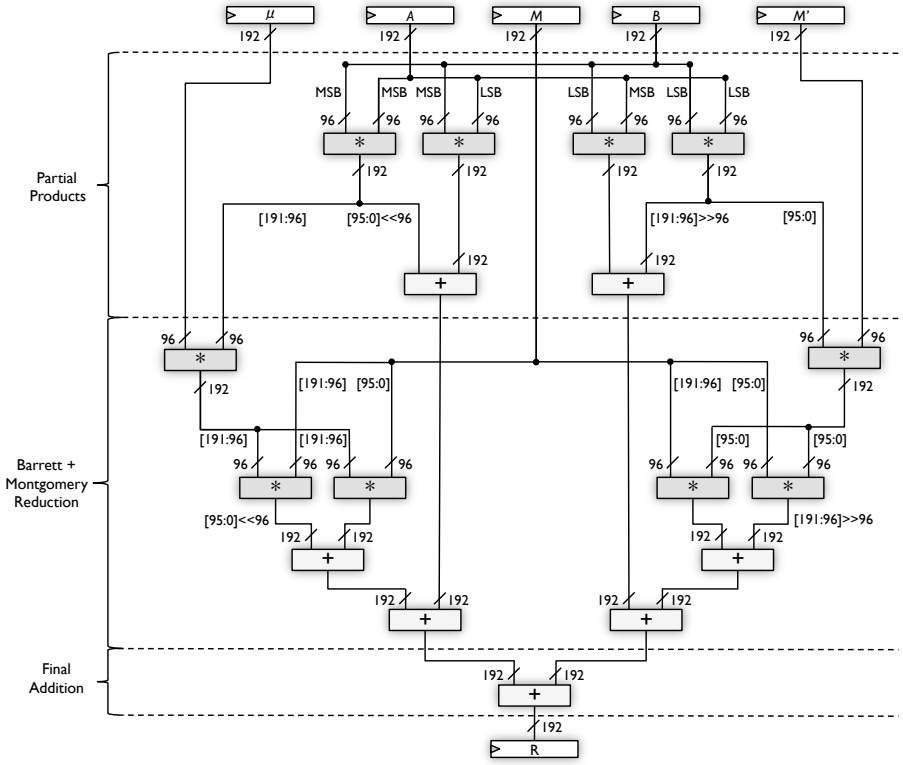

Fig. 5. Datapath of a fully-parallel modular multiplier based on bipartite algorithm.

in parallel. Assuming the reduction of $p_0R^{-1} \bmod M$ is done by the means of Montgomery's algorithm, we need three $\frac{n}{2} \times \frac{n}{2}$-bit multipliers. Similarly, for the reduction of $p_1R \bmod M$, which is now done by the means of Barrett's algorithm, we again need three $\frac{n}{2} \times \frac{n}{2}$-bit multipliers. Finally, for the reduction of $(p_2 - p_0 - p_1) \bmod M$, we need another three $\frac{n}{2} \times \frac{n}{2}$-bit multipliers, i.e. in order to calculate p_0, p_1, and p_2. To summarize, for the fully-parallel version of tripartite multiplier, we need nine $\frac{n}{2} \times \frac{n}{2}$-bit multipliers. The architecture implementing $n = 192$-bit tripartite modular multiplier is shown in Fig. 6.

To summarize, the fully-parallel implementation of our proposed algorithm requires only nine $\frac{n}{2} \times \frac{n}{2}$-bit multipliers, while the implementation based on bipartite requires ten. Implementations based on Barrett and Montgomery algorithms require twelve $\frac{n}{2} \times \frac{n}{2}$-bit multipliers. Therefore, while achieving the same speed, our multiplier requires considerably less area. To prove this in practice, we provide implementation results in Table 1.

Table 1. Comparison of ASIC implementations for a fully-parallel 192 × 192-bit modular multiplier. Target platform: UMC 0.13 μm CMOS technology (Synopsys Design Compiler version C-2009.06-SP3, synthesis results).

Algorithm	Area [kGE]	Number of cycles	Frequency [MHz]	T_r [MHz]	Efficiency [GE×ms]
Classical	383,992	1	5	5	76.8
	624,333	1	12.8	12.8	48.8
Montgomery	384,921	1	5	5	77.0
	598,872	1	25.6	25.6	23.4
Bipartite	361,788	1	5	5	72.4
	572,337	1	24.4	24.4	23.5
Proposed ($u = 2$)	324,569	1	5	5	64.9
	512,995	1	27.7	27.7	18.5

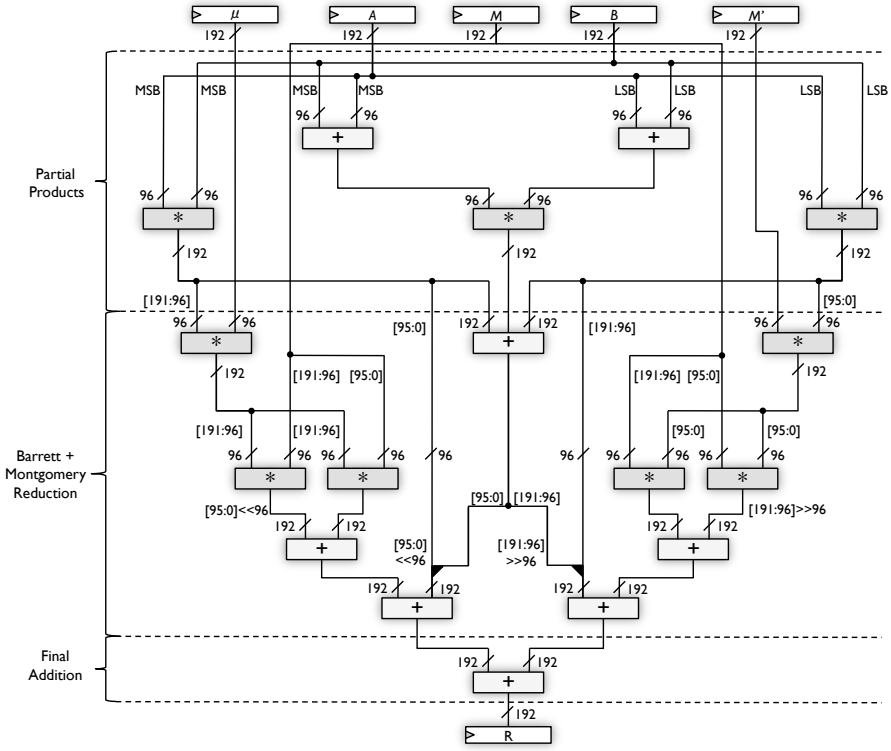

Fig. 6. Datapath of a fully-parallel modular multiplier based on the proposed algorithm.

To compare different architectures with respect to speed, we use a relative throughput defined as:

$$T_r = \frac{f}{N} \ , \tag{11}$$

where f is frequency and N represents the total number of cycles needed for one modular multiplication. Clearly, for the fully-parallel implementations it holds $T_r = f$.

As can be observed from Table 1, we first provide the synthesis results with the fixed frequency of 5 MHz and compare all the architectures with respect to area. It is obvious that our design consumes up to 15 % less area in this case. Furthermore, we synthesize all the architectures with the maximum achievable frequency and show that our design provides the fastest relative throughput of 27.7 MHz. In terms of efficiency, our design has a time-area product of only 18.5 GE×ms and clearly outperforms all other architectures.

Although running at very high speed, the fully parallel implementation comes at large area overhead. Therefore, we consider another possible approach by substantially decreasing the area overhead, yet maintaining the good speed performance.

Digit-Serial Implementation In order to have a fast multiplier while preserving a relatively low area, we consider here a digit-serial approach. We further assume that the multiplier has enough parallel processing units such that the full degree of parallelism can be exploited both for bipartite and tripartite algorithms.

To make a fair comparison of modular multipliers implemented using different algorithms we use the following criteria. A computational complexity of the algorithm is considered to be the number of SP multiplications necessary for performing a single modular multiplication. Therefore, we assume that an addition can be implemented very efficiently in hardware. Since the only requirement for achieving full parallelism is to use a sufficient number of SP multipliers, we consider only the number of SP multipliers as the area estimation. We stress here that the size of the whole architecture will, of course, be only proportional and not equal to the size of all SP multipliers. The size of a single-precision multiplier is $w \times w$ bits.

We assume that, besides maximizing the throughput, an important goal is to keep the number of SP multipliers as small as possible. Hence, we consider the case with the minimal number of SP multipliers while still allowing fully parallel processing. The critical path delay is assumed to be the same in all cases and equal to the delay of one SP multiplier – t_{SP}. Finally, the total speed-up is considered for the case when n_w is big. This is a theoretical result and will be different in practice as the final speed also depends on the physical size of a design.

We first calculate the computational complexities for the classical modular multiplication based on Barrett's algorithm, Montgomery multiplication and bipartite modular multiplication, respectively. We assume that an SP multiplication requires a single clock cycle, while the addition in hardware can be implemented within the same clock cycle at a reasonable hardware cost.

The complexity of the classical modular multiplication based on Barrett reduction can be simply obtained by analyzing Alg. 1. At step three of the algorithm we need to perform n_w SP multiplications. Using the trick of Barrett, step four can be performed at the cost of a single digit multiplication. Finally, step five of the algorithm requires another n_w digit multiplications. In total, to perform a classical modular multiplication, we need $2n_w^2 + n_w$ SP multiplications.

By analyzing Alg. 2 we conclude that the complexity of Montgomery multiplication is also equal to $2n_w^2 + n_w$ SP multiplications.

Assuming that we combine the classical modular multiplication based on Barrett reduction and Montgomery multiplication to implement the bipartite algorithm, the complexity becomes $n_w^2 + n_w/2$ SP multiplications (for the case of $k = \lceil n_w/2 \rceil$). Although it requires more resources, the bipartite algorithm can speed up a modular multiplication by up to two times.

Next, we provide the analysis of computational complexity for the proposed, tripartite algorithm. Let us consider the general case, namely a u-split version of the proposed algorithm. Assuming the algorithm being fully parallelized, the most time-consuming part is the classical modular multiplication based on Barrett reduction and the Montgomery multiplication. As discussed in Section 2, in order to compute the $n_w \times n_w$-digit modular multiplication, they use $\lambda = w + 4$-bit and $\lambda = w$-bit digit SP multiplications, respectively. Step three of both Alg. 1 and Alg. 2 requires n_w/u SP multiplications, while step four requires only one SP multiplication. Since the modulus M is an n_w-digit integer, step five of both algorithms still requires n_w SP multiplications. Finally, to perform a full $n_w \times n_w$-digit modular multiplication we need

$$\frac{n_w}{u}\left(\frac{n_w}{u} + 1 + n_w\right) = \frac{u+1}{u^2}n_w^2 + \frac{1}{u}n_w \tag{12}$$

single-precision multiplications.

Table 2. Cost and performance estimation for an interleaved digit-serial modular multiplication.

Algorithm	Number of SP multipliers†	Number of cycles	Critical path delay‡	Speed-Up (n_w is big)
Classical	1	$2n_w^2 + n_w$	t_{SP}	1
Montgomery	1	$2n_w^2 + n_w$	t_{SP}	1
Bipartite	2	$n_w^2 + \frac{1}{2}n_w$	t_{SP}	2
Proposed ($u = 2$)	3	$\frac{3}{4}n_w^2 + \frac{1}{2}n_w$	t_{SP}	2.67
Proposed ($u = 4$)	9	$\frac{5}{16}n_w^2 + \frac{1}{4}n_w$	t_{SP}	6.40
Proposed ($u = 8$)	27	$\frac{9}{64}n_w^2 + \frac{1}{8}n_w$	t_{SP}	14.22

†A single-precision multiplier is of size $w \times w$ bits.

‡A critical path delay is assumed to be the latency of a single-precision multiplier.

The data in Table 2 only represents the theoretical result based on the number of SP multipliers used in our design. We further compare the complexities of all the aforementioned algorithms by providing Fig. 7.

To show the practical value of our proposal, we implement an interleaved, digit-serial modular multiplier and compare it with the implementations of other algorithms. First, we provide the figures for the Xilinx Virtex-5 FPGA board (xc5vlx50t-ff1136) and second, we provide the ASIC synthesis results using UMC 0.13 μm CMOS process and Synopsys Design Compiler version C-2009.06-SP3. Additionally,

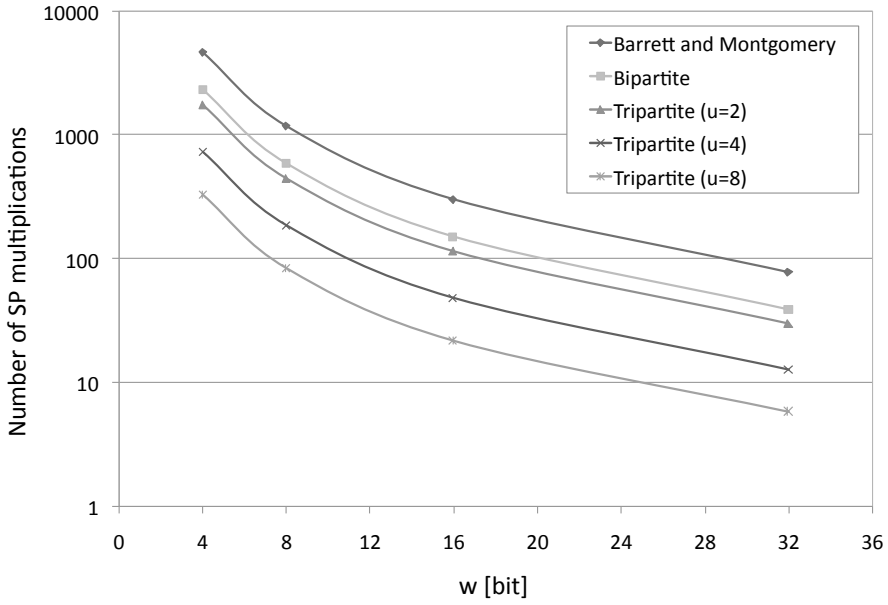

Fig. 7. Number of SP multiplications needed for one 192×192-bit modular multiplication as a function of different digit-sizes (w is ranging from 4 to 32).

we implement the classical multiplier based on Barrett reduction, the Montgomery and the bipartite multipliers, and compare them with our results.

For the FPGA implementation, our strategy is to use dedicated DSP48E slices on the board for implementing SP multipliers. Implemented this way, the whole design achieves a higher speed and consumes less bit-slices. However, DSP48E slices are a limited resource on our testing FPGA board and hence the goal is to use the minimal number of SP multipliers, yet allowing the full parallelism on the algorithmic level.

The described strategy results in an architecture for classical and Montgomery algorithms as shown in Fig. 8. As mentioned in Section 2, we use $\lambda = w$ bits digit-size for the case of Montgomery and $\lambda = w + 4$ bits digit-size for the case of classical modular multiplication based on Barrett reduction. A single $\lambda \times \lambda$ SP multiplier is used in both cases (inside the digit-serial multiplier π_1).

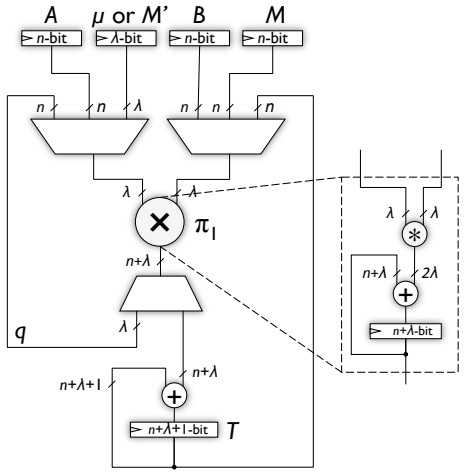

Fig. 8. Datapath of a digit-serial modular multiplier based on classical and Montgomery algorithms.

An architecture for the bipartite method is described in Fig. 9. It consists of a classical modular multiplier based on Barrett reduction and a Montgomery multiplier. Two digit-serial multipliers (π_1 and π_2) were used, each containing one SP multiplier.

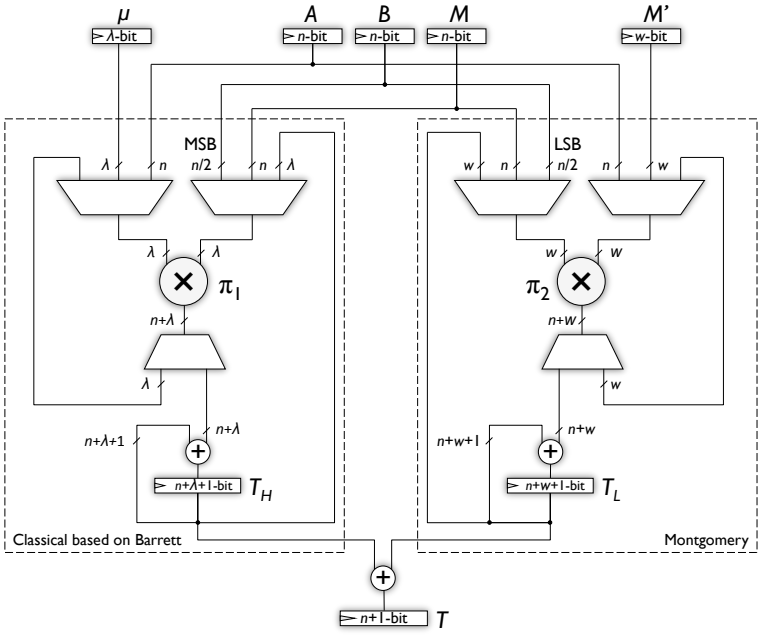

Fig. 9. Datapath of a digit-serial modular multiplier based on the bipartite algorithm.

Finally, our proposed architecture for the $u = 2$-split version is depicted in Fig. 10. As discussed in Section 3.1, it consists of a classical modular multiplier based on Barrett reduction (π_1), Montgomery multiplier (π_2) and Karatsuba multiplier in the middle (π_3) – all running in parallel. Instead of computing $A_1B_1R \bmod M$, the classical modular multiplier on the left-hand side of Fig. 10 computes $A_1B_1R \bmod M - A_1B_1$ and stores the result in register T_H where A_1B_1 is a partial product necessary for the Karatsuba part. The same holds for the Montgomery multiplier which instead of $A_0B_0R^{-1} \bmod M$ computes $A_0B_0R^{-1} \bmod M - A_0B_0$ and stores the result in register T_L. Finally, all three parts are added together, resulting in the final $n + 2$-bit product.

Since in most cryptographic applications the result of one modular multiplication is used as input to another, successive modular multiplication, the result needs to be reduced and remain strictly smaller than the modulus. At most three subtractions are necessary for this final reduction.

Table 3 shows the results of our FPGA implementations. We have implemented a 192×192-bit modular multiplier with digit size of $w = 32$ bits, based on all the mentioned algorithms and compared them with our solution. All the designs were implemented using maximum speed as the main constraint. The results clearly show that concerning speed, our algorithm outperforms all the previously proposed solutions. As the comparison criteria we use the relative throughput defined above.

An area overhead of about 660 bit-slices, compared to the bipartite design, is mainly due to the use of additional adders and multiplexers necessary for the control logic. A better resource sharing is also possible at the cost of maximum frequency and therefore we have obtained a design of 1408 bit-slices running at a frequency of 69 MHz. Finally, a speed-up of almost 150 % compared to the classical method and about 25 % compared to the bipartite method is obtained.

Table 4 shows the results of our ASIC implementations. The same architectures as in the case of FPGA were synthesized and compared to each other. As can be observed, concerning the speed performance, our proposal outperforms the classical modular multiplier by nearly 3 times and the bipartite multiplier by 45 %. However, this comes at a larger hardware cost and therefore our multiplier is around 2.38 times bigger than the classical one and about 56 % bigger than the bipartite multiplier. The efficiency of our proposal in terms of time-area product is less than the efficiency of Montgomery and Bipartite

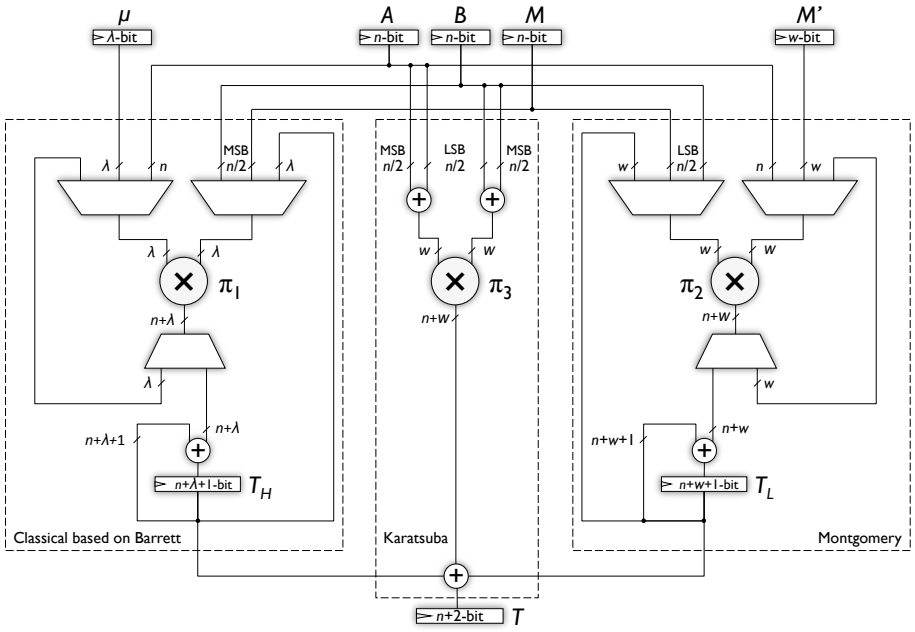

Fig. 10. Datapath of a digit-serial modular multiplier based on the proposed algorithm.

Table 3. Comparison of FPGA implementations for a digit-serial 192×192-bit modular multiplier. Target platform: Xilinx Virtex 5 FPGA board (xc5vlx50t-ff1136).

Algorithm	Bit slices	DSP48E slices	Number of cycles	Frequency [MHz]	T_r [MHz]	Speed Up
Classical	988	6	78	77	0.987	1
Montgomery	849	4	78	102	1.308	1.325
Bipartite	1313	10	39	77	1.974	2
Proposed ($u = 2$)	1979	14	30	74	2.467	2.499

multiplier. We can also observe that the efficiency of Bipartite multiplier is smaller than the efficiency of Montgomery's. This indeed shows that both bipartite and tripartite algorithms are designed to provide fast modular multiplication by exploiting the parallelism induced by their own design goals.

5 Conclusions and Future Work

We have introduced a new modular multiplication algorithm suitable for efficient hardware implementations. We have also provided a proof-of-concept hardware modular multiplier based on the proposed algorithm that effectively integrates three different algorithms, the classical modular multiplication based on Barrett reduction, Montgomery multiplication and Karatsuba multiplication. The results show that concerning the speed, our proposed algorithm outperforms all the previously proposed solutions. Moreover, the fully-parallel implementation of our algorithm consumes less area and is more efficient than its counterparts.

We believe that the proposed algorithm offers a new alternative for efficient modular multiplication on both software and hardware platforms. The cost and performance trade-offs when increasing the value of u further (i.e. $u > 8$) still need to be explored. The software implementation and the scheduling problem on multicore platforms still remains a challenge. We also plan to implement the proposed algorithm for multiplication in binary field. Another direction for future work would be to use the Toom-Cook algorithm for the multiplication step instead of Karatsuba's method.

Table 4. Comparison of ASIC implementations for a digit-serial 192×192-bit modular multiplier. Target platform: UMC 0.13 μm CMOS technology (Synopsys Design Compiler version C-2009.06-SP3, synthesis results).

Algorithm	Area [GE]	Number of cycles	Frequency [MHz]	T_r [MHz]	Speed Up	Efficiency [GE×ms]
Classical	42,331	78	191	2.448	1	17.3
Montgomery	31,704	78	229	2.936	1.199	10.8
Bipartite	64,450	39	193	4.949	2.022	13.0
Proposed ($u = 2$)	100,771	30	215	7.167	2.928	14.06

Acknowledgments

This work is supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State, by FWO project G.0300.07, by the European Commission under contract number ICT-2007-216676 ECRYPT NoE phase II, and by K.U.Leuven GOA TENSE number BOF GOA/11/.

References

1. P. Barrett. *Communications Authentication and Security using Public Key Encryption - A Design for Implementation*. Master Thesis, Oxford University, 1984.
2. P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Proc. CRYPTO'86*, pages 311–323, 1986.
3. J.-F. Dhem. *Design of an Efficient Public-key Cryptographic Library for RISC-based Smart Cards*. PhD Thesis, 1998.
4. J. Fan, K. Sakiyama, and I. Verbauwhede. Montgomery Modular Multiplication Algorithm on Multi-Core Systems. In *IEEE Workshop on Signal Processing Systems: Design and Implementation (SIPS 2007)*, pages 261–266, Shanghai,China, 2007. IEEE.
5. M. E. Kaihara and N. Takagi. Bipartite Modular Multiplication. In J. R. Rao and B. Sunar, editors, *Proceedings of 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 3659 in Lecture Notes in Computer Science. Springer-Verlag, 2005.
6. A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Translation in Physics-Doklady*, 145:595–596, 7 1963.
7. N. Koblitz. Elliptic Curve Cryptosystem. *Math. Comp.*, 48:203–209, 1987.
8. V. Miller. Uses of Elliptic Curves in Cryptography. In H. C. Williams, editor, *Advances in Cryptology: Proceedings of CRYPTO'85*, number 218 in Lecture Notes in Computer Science, pages 417–426. Springer-Verlag, 1985.
9. P. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985.
10. M. J. Potgieter and B. J. van Dyk. Two Hardware Implementations of the Group Operations Necessary for Implementing an Elliptic Curve Cryptosystem over a Characteristic Two Finite Field. In *Africon Conference in Africa, 2002. IEEE AFRICON. 6th*, pages 187–192, 2002.
11. R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
12. Daisuke Suzuki. How to Maximize the Potential of FPGA Resources for Modular Exponentiation. In *CHES '07: Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, pages 272–288, Berlin, Heidelberg, 2007. Springer-Verlag.
13. A. Toom. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Translations of Dokl. Adad. Nauk. SSSR*, 3, 1963.
14. H. Wu. Montgomery Multiplier and Squarer for a Class of Finite Fields. *IEEE Transactions on Computers*, 51(5):521–529, May 2002.